Università di Pisa

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laurea Triennale in Ingegneria Informatica

Implementazione del sistema di protezione contro Meltdown nel nucleo didattico

Candidato:
Riccardo Sagramoni
Matricola 565472

Relatore:
Ing. Giuseppe Lettieri

Anno Accademico 2019-2020

Indice

Capitolo 1

Meltdown e il sistema di protezione KAISER

La sicurezza dei sistemi informatici attuali si fonda sull'isolamento della memoria, ad esempio marcando come privilegiati gli indirizzi di memoria kernel e bloccando eventuali accessi da parte di programmi utente [14]. **Meltdown** è un tipo di attacco informatico che sfrutta un effetto collaterale dell'esecuzione fuori ordine nei processori moderni per leggere locazioni di memoria scelte in maniera arbitraria. L'attacco funziona su varie microarchitetture Intel prodotte sin dal 2010, indipendentemente dal sistema operativo in uso. Meltdown è quindi in grado di accedere arbitrariamente a qualsiasi locazione di memoria protetta (afferenti al kernel o ad altri processi) senza necessitare alcun permesso o privilegio da parte del sistema [16].

Meltdown rompe quindi tutti i meccanismi di sicurezza che si basano sull'isolamento degli spazi di indirizzamento, andando a colpire milioni di utenti. Il sistema di protezione KAISER, sviluppato originariamente per KASLR [3], ha l'importante effetto secondario di impedire l'utilizzo di Meltdown [16].

L'obiettivo di questa tesi è stato quello di proteggere da Meltdown il nucleo didattico utilizzato nel corso di "Calcolatori Elettronici" tenuto dall'Ing. Giuseppe Lettieri, implementando una versione ottimizzata di KAISER.

Nella nostra trattazione, mostreremo prima le caratteristiche e le cause dell'attacco Meltdown e in che modo il sistema KAISER possa proteggere il kernel da questo tipo di attacco (capitolo 1). Dopodiché verranno descritte le caratteristiche peculiari del nucleo didattico (capitolo 2 a pagina 10). Infine, concluderemo con la presentazione delle modifiche apportate al nucleo (capitolo 3 a pagina 14) e con il codice prodotto (capitolo 4 a pagina 18).

1.1 Background

Presenteremo ora sinteticamente tre concetti alla base dell'attacco Meltdown: l'esecuzione fuori ordine, gli spazi di indirizzamento e gli attacchi alla memoria cache.

1.1.1 Esecuzione Fuori Ordine

L'esecuzione fuori ordine è una tecnica di ottimizzazione che permette di massimizzare l'utilizzo delle unità di esecuzione della CPU [16]. Ogni istruzione Assembly viene prelevata dalla memoria e poi decodificata dalla CPU, ovvero viene tradotta in una o più *micro-operazioni*. La CPU non segue il flusso lineare di istruzioni, ma esegue ogni micro-operazione *non appena tutte le risorse necessarie sono disponibili* (che siano risorse hardware o dati restituiti da altre operazioni). Si dice quindi che la CPU non esegue le istruzioni linearmente, ma **in maniera speculativa** [1].

Le micro-operazioni vengono inserite in una coda di riordino nell'ordine previsto dal programma. Gli effetti di una micro-operazione completata senza errori possono essere applicati sulla RAM e sui registri visibili quando non vi sono altre micro-operazioni davanti nella coda. Si dice, in questo caso, che la micro-operazione (o in senso più largo, l'istruzione) viene *ritirata* [1]. Si noti che le istruzioni vengono ritirate dalla CPU nell'ordine in cui sarebbero state eseguite se la CPU avesse seguito in maniera lineare il flusso d'istruzioni.

Nell'esecuzione fuori ordine, è comune che vengano eseguite (ma non ritirate) alcune istruzioni che non fanno parte del flusso lineare di controllo [1], dette *transient instruction* [16]. Questo è dovuto al fatto che la CPU cerca di "indovinare" la direzione che intraprende il flusso di istruzioni in corrispondenza di una istruzione di salto, sfruttando tecniche predittive che variano a seconda dell'hardware [1].

Nel caso di una predizione di salto *errata*, le istruzioni *non* vengono ritirate e gli effetti vengono annullati (*rollback delle istruzioni*). In questo modo, le transient instruction non hanno alcun effetto sulla macroarchitettura (memoria centrale e registri visibili del processore).

1.1.2 Spazi di indirizzamento

Per risolvere diversi problemi, in particolare l'isolamento dei processi [11], le CPU supportano l'utilizzo di spazi d'indirizzamento virtuali, in cui gli indirizzi virtuali (relativi al singolo processo) vengono tradotti in indirizzi fisici. Lo spazio d'indirizzamento di un processo (ovvero tutti i possibili indirizzi che un processo può generare) viene suddiviso in regioni dette *pagine* che possono essere mappate individualmente nella memoria fisica attraverso una tabella di traduzione multivello [15]. Ogni processo possiede una propria

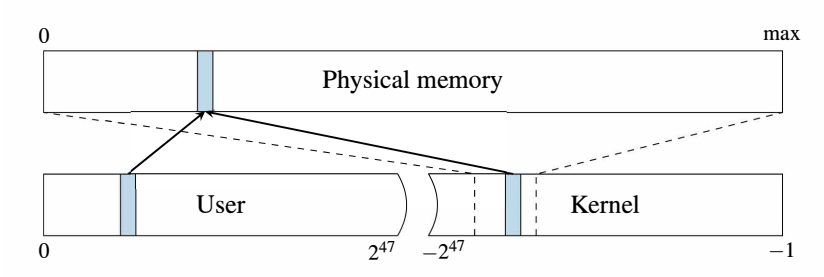


Figura 1.1: Ogni indirizzo fisico del processo è mappato sia nello spazio d'indirizzamento utente che in quello kernel all'interno della finestra di memoria fisica

tabella di traduzione che traduce tutti e soli i suoi indirizzi virtuali e che definisce le proprietà di protezione delle varie zone di memoria.

Per permettere ad ogni processo di usufruire delle primitive di sistema offerte dal kernel e alle routine di sistema di accedere liberamente all'intera memoria fisica (ad esempio per modificare le tabelle di traduzione di un processo), si utilizza una traduzione, denominata *finestra di memoria fisica*, che mappi l'intera memoria fisica, compresi il kernel e lo spazio di memoria di tutti i processi, nello spazio di indirizzamento accessibile solo da livello privilegiato (figura 1.1) [13].

1.1.3 Attacchi Cache

Al fine di velocizzare gli accessi alla RAM, le CPU contengono buffer di memoria molto veloce ma di dimensioni limitate che costituiscono la cosiddetta *memoria cache*. La memoria cache maschera i tempi di latenza estremamente lunghi per l'accesso alla memoria centrale (molto lenta in confronto alla cache) conservando le locazioni di memoria che, secondo principi statistici come la *località spaziale* (se un programma accede ad un certo indirizzo, è molto probabile che in breve tempo accederà ad un indirizzo vicino) e la *località temporale* (se un programma accede ad un certo indirizzo, è molto probabile che in breve tempo vi accederà di nuovo), è più probabile vengano indirizzate dalla CPU nel breve periodo [10].

Gli attacchi a canale laterale (*side-channel attacks*) contro la cache sfruttano questa differenza di tempo di accesso introdotta dalla cache stessa. Negli attacchi Flush+Reload [17], usati da Meltdown [16], l'attaccante è in grado di determinare se una locazione di memoria è stata precedentemente caricata in cache, misurando il tempo impiegato da un'operazione di lettura.

1.2 Come agisce Meltdown

L'attacco Meltdown consiste in tre passi fondamentali [16]:

1. Leggere il contenuto di una locazione di memoria inaccessibile dall'attaccante, causando il lancio di un'eccezione di protezione

2. Accedere in maniera speculativa ad una linea di memoria cache in base al contenuto segreto della locazione protetta

3. Usare un'attacco di tipo Flush+Reload per determinare il contenuto segreto in base a quale linea di memoria è stata acceduta

1.2.1 Passo 1: Leggere il segreto

Nel prima passo di Meltdown, l'attaccante cerca di accedere ad una zona di memoria protetta, ad esempio la memoria kernel. Il tentativo di accesso ad una pagina non accessibile da livello utente fa in modo che la CPU sollevi un'eccezione di protezione, che generalmente termina il processo. Tuttavia, a causa dell'esecuzione fuori ordine, la CPU potrebbe aver già eseguito l'istruzione di accesso in maniera speculativa *prima* delle istruzioni relative all'eccezione di protezione, al fine di minimizzare i tempi di latenza (vedi paragrafo 1.1.1). In questo modo la CPU accederebbe in maniera speculativa alla locazione di memoria desiderata prima che il processo venga terminato.

Grazie al lancio dell'eccezione, le eventuali istruzioni eseguite in maniera speculativa (le *transient instruction*), che non sarebbero dovute essere eseguite in quanto relative ad una previsione di salto *errata*, non vengono *ritirate* dalla CPU e non hanno così alcun effetto sulla macroarchitettura in generale (memoria centrale e registri logici non speculativi del processore) [1].

1.2.2 Passo 2: Trasmettere il segreto

Per poter trasmettere il segreto, si utilizza un *probe array*, di dimensione pari a 256 pagine virtuali e allocato precedentemente nella memoria del processo attaccante, assicurandosi che *nessuna porzione dell'array sia presente nella cache*. La sequenza di transient instruction contiene un accesso ad un elemento del probe array, il cui offset è calcolato moltiplicando il valore del byte per la dimensione di una pagina virtuale (tipicamente e nel nostro sistema è *4KiB* [11]).

Quando la CPU gestisce l'eccezione di protezione causata dal Meltdown, le transient instruction non vengono ritirate dalla CPU, senza avere dunque effetti a livello di macroarchitettura. Sebbene quindi non sia possibile rendere direttamente disponibile il segreto dal programma utente, si hanno importanti effetti secondari a livello di **microarchitettura**, in particolare nella memoria cache [16].

Durante l'esecuzione speculativa, infatti, la locazione di memoria all'interno del probe array che viene acceduta dalla CPU, viene memorizzata in memoria cache e

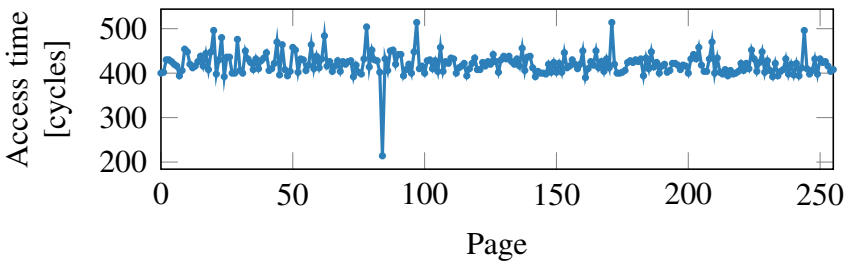


Figura 1.2: Tempo di accesso alle 256 pagine del probe array. Il grafico mostra una *cache hit* sulla pagina acceduta nel passo 2. [16]

vi rimane anche in seguito all'annullamento degli effetti delle transient instruction, rendendola vulnerabile ad un attacco side-channel.

L'utilizzo del valore segreto moltiplicato per la dimensione della pagina, ci garantisce sia una precisa *correlazione* tra il valore segreto e la locazione caricata in memoria, sia che a differenti valori della locazione di memoria saranno accedute differenti *pagine* del probe array. Ciò previene il fatto che il prefetcher hardware (per ragioni di ottimizzazione) potrebbe caricare in cache anche le locazioni di memoria adiacenti a quella acceduta, rendendo impossibile determinare quale locazione di memoria sarebbe stata indirizzata se non fosse stato utilizzato a priori questo accorgimento.

1.2.3 Passo 3: Ricevere il segreto

Dopo che la sequenza di istruzioni del passo 2 è stata eseguita, in cache è presente esattamente una linea di memoria del probe array. L'offset di questa linea è dipendente esclusivamente dal valore segreto presente nell'arbitaria locazione di memoria protetta. Grazie a ciò, l'attaccante può effettuare un'attacco Flush+Reload [17], iterando attraverso le 256 pagine del probe array e misurando il tempo di accesso per il primo elemento di ogni pagina (vedi figura 1.2). In base a quanto detto finora, la pagina con la latenza minore è l'unica presente in memoria cache e il numero della pagina è il *valore segreto letto dalla memoria protetta.*

1.2.4 Conclusioni

Il seguente codice mostra in Assembly x86-64 la sequenza di istruzioni alla base di Meltdown, relative ai passi 1 e 2 dell'attacco.

```
1 # rcx = indirizzo di memoria kernel
2 # rbx = probe array
3 movb (%rcx), %al          # Lettura del segreto
4 shl  $12, %rax            # Traslazione dell'offset
5 movq (%rbx, %rax), %rbx   # Trasmissione del segreto
```

Meltdown è quindi in grado di leggere in maniera arbitraria dati presenti in memoria *protetta*, ad esempio nello spazio di indirizzamento kernel. L'efficienza di Meltdown si basa principalmente sull'esistente *race condition* tra il lancio dell'eccezione di protezione e il passo 2 del nostro attacco (vedi paragrafo 1.2.2 a pagina 6). Ovvero le transient instruction devono essere eseguite speculativamente *prima* dell'handler dell'eccezione. Per questo motivo, sono previsti alcuni accorgimenti e ottimizzazioni ulteriori non significativi per la nostra trattazione e per le quali rimandiamo a Lipp et al. [16].

Dato che l'intera memoria fisica viene mappata all'interno dello spazio di indirizzamento del kernel attraverso la cosiddetta *finestra di memoria fisica* [13], Meltdown è in grado non solo di leggere le zone di memoria relative al kernel, ma anche gli spazi di memoria di tutti gli altri processi. In base a quanto rilevato da Lipp et al. [16], Meltdown è in grado di effettuare il dump dell'intera memoria fisica fino ad una velocità di 503 KB/s.

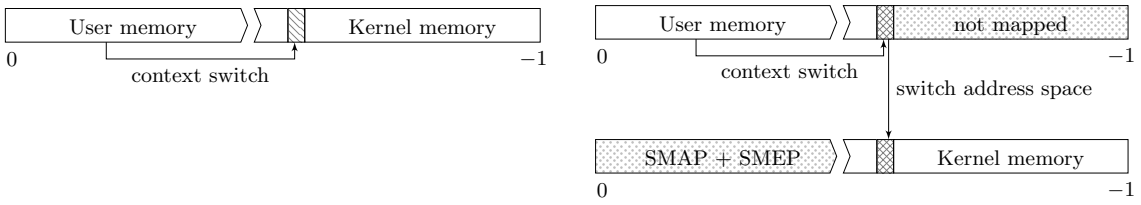
1.3 Il sistema di protezione KAISER

KAISER, proposto da Gruss et al. [3], è una modifica del nucleo in cui il kernel non viene mappato nello spazio virtuale dei processi utente. Questa modifica era stata pensata per prevenire attacchi side-channel contro la misura di protezione KASLR [5, 4, 6], ma ha l'importante effetto secondario di prevenire Meltdown [16].

L'idea alla base di KAISER è quella di separare lo spazio di memoria kernel da quello utente, ovvero di rendere disponibile la traduzione degli indirizzi virtuali del kernel *soltanto* quando il sistema si trova in modalità privilegiata. Ciò previene Meltdown in quanto, quando il sistema sta eseguendo il programma utente della sezione 1.2.4 nella pagina precedente, l'indirizzo virtuale scelto non è presente nell'albero di traduzione del processo attaccante e quindi la CPU non è in grado di accedervi neanche in maniera speculativa. Il meccanismo proposto da Gruss et al. è quindi ritenuto la miglior soluzione a breve termine per proteggere i sistemi informatici da Meldown [16].

KAISER introduce il concetto di **spazi d'indirizzamento shadow** per garantire l'isolamento della memoria kernel. Come mostrato in figura 1.3 nella pagina seguente, ogni processo possiede due spazi di indirizzi: uno spazio d'indirizzamento shadow, in cui è mappato solo lo spazio di memoria utente e una porzione del kernel necessaria per le interruzioni, e uno spazio d'indirizzamento in cui è mappato sia l'intero kernel che lo spazio utente (in figura lo spazio utente è protetto con Supervisor Mode Access Prevention e Supervisor Mode Execution Prevention per compatibilità con x86 Linux) [3].

Ogni qualvolta che il programma passerà da livello utente a livello sistema (ad esempio attraverso una primitiva di sistema o il lancio di un'interruzione) e viceversa, la CPU dovrà aggiornare il registro **CR3** con il valore della tabella di livello 4 corrispondente al nuovo livello di privilegio (tabella shadow per il livello utente e tabella kernel



(a) Regular OS (b) KAISER

Figura 1.3: (a) Il kernel è mappato nella memoria virtuale di ogni processo. (b) Creazione dello spazio di indirizzamento shadow senza il kernel (solo alcune porzioni vengono mantenute per la gestione delle interruzioni)

per il livello sistema). Sarà dunque necessario implementare una o più funzioni (le *funzioni trampolino*), mappate in memoria shadow, dedicate all'aggiornamento corretto del registro CR3 ad ogni modifica del livello di privilegio.

Come accennato sopra, nell'architettura x86 sono necessarie alcune porzioni del kernel per il corretto funzionamento delle interruzioni e devono perciò essere mappate nello spazio di indirizzamento shadow:

- la Interrupt Descriptor Table (IDT);
- la Global Descriptor Table (GDT);
- i Task State Segment (TSS);
- la pila sistema del processo;
- la pila e il vettore delle richieste di interruzioni;
- il codice di entrata e uscita dagli interrupt handler.

Capitolo 2

Introduzione al nucleo didattico

Il nucleo didattico è un **kernel a 64 bit** perfettamente funzionante, utilizzato per finalità didattiche nel corso di Calcolatori Elettronici di Ingegneria Informatica presso l'Università di Pisa e sviluppato a partire dai concetti presentati in *Architettura dei calcolatori Vol. III* di Frosini e Lettieri[2].

Il sistema è organizzato in tre moduli distinti [8]:

- **SISTEMA**, eseguito con il processore a livello sistema, che contiene la realizzazione dei processi, inclusa la gestione della memoria;
- **I/O**, eseguito con il processore a livello sistema, che contiene le routine di ingresso/uscita che permettono di utilizzare le periferiche collegate al sistema;
- **UTENTE**, eseguito con il processore a livello utente, che contiene il programma che il nucleo dovrà eseguire.

I moduli sistema e I/O forniscono un supporto al modulo utente, sotto forma di *primitive* che questo può invocare.

Il sistema sviluppato, per quanto funzionante, non è autosufficiente e per sviluppare i moduli necessita di un altro sistema di appoggio, nel nostro caso Linux, i cui strumenti devono essere opportunamente configurati in modo che produca eseguibili per il nostro sistema. Il nucleo così sviluppato può essere eseguito sia su una macchina reale (sconsigliato), sia su un emulatore. Nel nostro caso, useremo una versione di QEMU opportunamente modificata [9].

Il modulo sistema deve essere caricato da un *bootstrap loader* mentre il modulo I/O e utente devono essere caricati da una *partizione di swap*, nel nostro caso emulata da un file binario

2.1 Gestione dei processi

All'interno del nucleo didattico, i processi vengono rappresentati attraverso due strutture dati:

- Il descrittore di processo (`des_proc`), contenente il TSS del processo e i valori dei registri salvati all'ultimo cambio di contesto. Viene indirizzato dalla entrata della GDT relativa al processo.
- Il `proc_elem`, contenente l'id e la priorità del processo e usato nelle code di processi.

Il sistema prevede una politica di schedulazione a priorità fissa, in cui passa in esecuzione il processo pronto con priorità *maggiore*. A parità di priorità, viene adottata una politica FIFO (*First Input, First Output*).

2.2 Gestione della memoria virtuale

Il nostro sistema implementa la paginazione su domanda [12], con zone di memoria condivise tra tutti i processi e zone private ai processi [7].

La memoria virtuale di ogni processo è implementata attraverso una tabella di traduzione multi-livello [15] ed è divisa nelle seguenti sezioni (vedi anche figura 2.1):

- Il sottospazio canonico superiore, con indirizzi da 0x0000000000000000 a 0x00007FFFFFFFFFFF, accessibile solo da livello sistema. A sua volta suddiviso in:
  - **sistema/condivisa**: contiene la finestra di memoria fisica
  - **sistema/privata**: contiene la pila sistema del processo
  - **IO/condivisa**: contiene il modulo I/O
- Il sottospazio canonico inferiore, con indirizzi da 0xFFFF800000000000 a 0xFFFFFFFFFFFFFFFF, accessibile sia da livello sistema che da livello utente. A sua volta suddiviso in:
  - **utente/condivisa**: contiene il modulo utente, ovvero le sezioni `.text` e `.data` del programma utente
  - **utente/privata**: contiene la pila utente del processo

Si noti come alcune parti della memoria fisica siano accessibili esclusivamente tramite la finestra: il primo MiB di memoria (riservato per ragioni storiche), lo heap di sistema, il modulo sistema, i descrittori di frame e i descrittori di pagine virtuali.

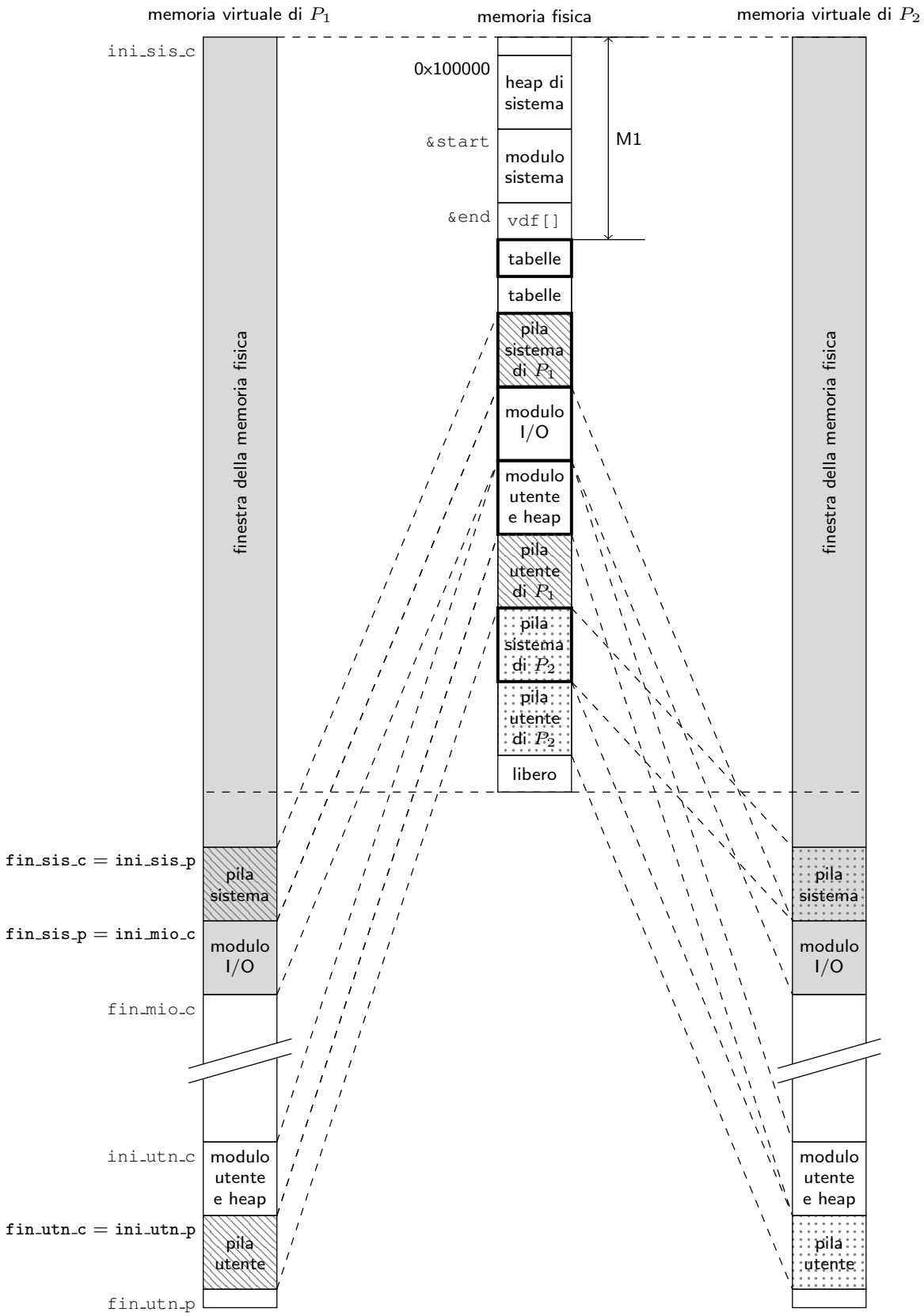


Figura 2.1: Esempio di memoria virtuale con due processi (non in scala) [7]

Usando la paginazione su domanda, la memoria fisica funziona da cache dello swap. Normalmente le sole sezioni *residenti*, ovvero non selezionabili come vittime per uno swap, sarebbero la *sistema/condivisa* e la *sistema/privata*, in quanto contenenti la finestra di memoria e la pila usata dal modulo sistema. Per semplicità di implementazione, nel nostro sistema sono state rese residenti tutte le sezioni condivise [7].

Capitolo 3

Implementazione del sistema di protezione

Nello svolgimento di questa tesi, abbiamo implementato sul nostro sistema una versione modificata di KAISER per proteggere il nucleo da Meltdown. Per semplicità, è stato protetta soltanto la sezione *sistema/condivisa* della memoria virtuale, contenente la finestra di memoria virtuale (vedi sezione 2.2 a pagina 11). Le sezioni io/condivisa e sistema/privata sono dunque ancora vulnerabili a Meltdown.

3.1 La finestra di memoria fisica

Nel nostro sistema, la finestra di memoria virtuale occupa interamente la *prima* entrata della tabella di livello 4 di ogni processo ed è inizializzata dal boot loader all'avvio della macchina virtuale. Grazie a questa proprietà, abbiamo potuto effettuare una prima ottimizzazione rispetto alla versione di KAISER proposta da Gruss et al. [3]: invece di creare lo spazio d'indirizzamento shadow a partire dalla tabella di livello 4, nel nostro sistema viene costruita a partire dalla *tabella di livello 3*. Al momento del passaggio nelle funzioni trampolino, il sistema modificherà la prima entrata della tabella di livello 4 del processo in esecuzione invece del registro CR4, inserendovi l'indirizzo della tabella di livello 3 "kernel" (se il processore sta passando a livello sistema) o "shadow" (se sta tornando a livello utente).

Oltre al risparmio di spazio per avere una tabella duplicata in meno, questa ottimizzazione evita lo svuotamento implicito del *Translation Lookaside Buffer* (TLB) dovuto alla modifica del registro CR4 [3], che avrebbe avuto un impatto negativo sulle prestazioni. Nella proposta di Gruss et al. [3], questo problema veniva risolto sfruttando alcune funzionalità delle CPU moderne di cui non disponiamo nel nostro sistema emulato.

3.2 La memoria kernel nello spazio shadow

Nel paragrafo 1.3 a pagina 8 abbiamo affermato che alcune porzioni del kernel devono essere mappate nello spazio di indirizzamento shadow per permettere il funzionamento delle interruzioni. Nella nostra implementazione le porzioni necessarie (che abbiamo denominato nel loro complesso come *memoria trampolino*) sono state raccolte in tre sezioni Assembly: una contenente il codice (`.trampoline_text`) e due contenenti i dati (`.trampoline_data` per le variabili non costanti e `.trampoline_bss` per quelle costanti). Ogni sezione è stato allineata alla dimensione delle pagine virtuali (4KiB), in modo che esse occupino per intero le pagine in cui sono allocate, evitando così la presenza di altre porzioni del kernel non determinabili a priori che, nel caso, sarebbero vulnerabili a Meltdown. Questo ci permette di inserirne la traduzione da indirizzo virtuale a fisico nello spazio di indirizzamento *shadow*.

3.3 Costruzione dello spazio di memoria shadow

Il sottoalbero di traduzione dello spazio di memoria shadow viene costruito durante l'inizializzazione del kernel dalla funzione `crea_finestra_FM_shadow` (riga 882 del listato 4.2 a pagina 19), che mappa in una tabella di livello 3 creata oppositamente (la tabella *shadow*) tutti gli indirizzi della *memoria trampolino*, ovvero le tre sezioni text, data e bss che costituiscono la porzione di kernel essenziale per le interruzioni.

Il meccanismo di mappatura delle pagine cosiddette *shadow* è identico a quello delle pagine normali nella memoria virtuale, ovvero si genera un albero di traduzione assicurandosi la presenza di tutte le tabelle necessarie ai vari livelli per tradurre gli indirizzi della *memoria trampolino* e marcando come assenti le altre pagine. Ciò impedirà a Meltdown di accedere al kernel e all'intera memoria fisica.

3.4 Le funzioni trampolino

Le funzioni trampolino d'ingresso (nel kernel) e di uscita (dal kernel) si occupano di aggiornare la prima entrata della tabella di livello 4 con il descrittore di tabella 3 opportuno. Mentre la tabella di livello 3 *kernel* viene creata dal boot loader, la tabella di livello 3 *shadow* viene creata dal nostro programma durante la fase di inizializzazione della memoria virtuale e il suo descrittore viene conservato nella variabile globale `des_finestra_shadow` (riga 754 del listato 4.2 a pagina 19). La sostituzione dello spazio di indirizzamento consiste in scrivere nella prima entrata della tabella di livello 4 del processo in esecuzione del contenuto di `des_finestra_shadow`, se stiamo passando al livello utente, o di `des_finestra_kernel` (inizializzato da noi; riga 1725 del listato 4.10 a pagina 39), se stiamo passando al livello sistema.

Sono state implementate due versioni delle funzioni trampolino:

- Le routine `trampoline_in` e `trampoline_out`, chiamate dal modulo sistema, che controllano se è necessario effettuare la sostituzione della finestra di memoria fisica attuale (ovvero se stiamo passando da livello utente a sistema o viceversa).

  Questo viene valutato dall'indirizzo salvato in pila durante il lancio dell'interruzione, che rappresenta l'indirizzo dell'ultima istruzione eseguita: se l'indirizzo appartiene al sottospazio inferiore della memoria virtuale, si può concludere che il flusso di controllo è passato dal modulo utente al modulo sistema (nel caso di trampolino di ingresso; riga 585 del listato 4.9 a pagina 34) o che tornerà nel modulo utente dopo l'esecuzione di `iretq` (nel caso di trampolino di uscita; riga 615 del listato 4.9 a pagina 34). In tal caso, è quindi necessario effettuare l'opportuna modifica alla finestra di memoria.

- Le primitive di sistema `a_trampoline_in` e `a_trampoline_out`, offerte al modulo I/O per effettuare manualmente il trampolino, che non richiedono il controllo dell'indirizzo in quanto non possono essere chiamate dal modulo utente (rispettivamente righe 634 e 658 del listato 4.9 a pagina 34).

La parte in comune tra le due versioni è stata accorpata in due funzioni chiamate `trampoline_in_2` e `trampoline_out_2` (righe 529 e 549 listato 4.9 a pagina 34).

L'utilizzo corretto delle funzioni trampolino è mostrato negli esempi 4.10 a pagina 39 (nel modulo sistema) e 4.12 a pagina 41 (nel modulo I/O).

3.5 La gestione dei TSS

Nel nostro sistema, ogni processo ha un proprio **Task State Segment** (TSS) all'interno della struttura `des_proc`, allocata nello heap di sistema quando il processo viene creato. Questa proprietà rende impossibile proteggere completamente il nucleo, in quanto i TSS sono *necessari* per le interruzioni e se mappassimo nello spazio shadow le pagine dello heap che contengono un TSS renderemmo vulnerabili a Meltdown gli altri dati allocati nella medesima pagina. Ciò comprometterebbe l'efficacia della protezione contro Meltdown.

Per questo motivo, abbiamo effettuato una separazione netta tra la componente del `des_proc` richiesta dall'hardware (il TSS) e la componente utilizzata lato software (in cui si trova il contesto salvato).

I TSS, essendo necessari nel nucleo didattico solo per l'indirizzo della pila sistema del processo in quanto la gestione dei processi è affidata al software, vengono accorpati in unico TSS, inizializzato dalla funzione `init_tss` (riga 1845 del listato 4.5 a pagina 26)

e allocato nella sezione `.trampoline_data` (riga 1721 del listato 4.11 a pagina 39), così da essere mappato nello spazio di memoria shadow. Al momento di caricare lo stato del processo in esecuzione, la routine `carica_stato` assegnerà al campo `punt_nucleo` del TSS l'indirizzo della base della pila sistema del processo (riga 133 del listato 4.6 a pagina 29), conservato nella nuova struttura `des_proc` (riga 40 del listato 4.1 nella pagina successiva).

Essendovi un solo TSS hardware, la funzione della sezione dei descrittori TSS della **Global Descriptor Table** (GDT) rimane solo quella di assegnare un id libero ai nuovi processi, in quanto tutte le entrate dei descrittori saranno uguali. Questa funzionalità può essere gestita in C++, creando un'array di puntatori a `des_proc` (la nuova struttura senza TSS) che assolverà alla duplice funzione di restituire un descrittore di processo dato l'id e gestire l'assegnazione e la rimozione degli identificatori dei processi (riga 47 del listato 4.1). La sezione dei descrittori TSS della GDT può essere compressa in una sola entrata (riga 1717 del listato 4.11 a pagina 39) e il registro TR (che contiene l'entrata della GDT che punta al TSS del processo in esecuzione) viene settato una sola volta in fase di inizializzazione del kernel (funzione `init_gdt`, riga 368, listato 4.8 a pagina 33).

3.6 Il TLB

Nonostante si sia evitato lo svuotamento implicito del TLB, è in ogni caso necessario invalidarlo forzatamente quando il sistema passa da sistema a utente. L'efficacia di KAISER si basa sulla garanzia che nell'albero di traduzione di ogni processo non vi sia il kernel e la finestra di memoria fisica (o la struttura equivalente per lo specifico sistema operativo) quando il processore lavora a livello utente e che *la CPU non abbia nessun altro modo per ottenere le traduzioni degli indirizzi.* Quando il processore lavora in modalità privilegiata, può accedere all'albero di traduzione completo di finestra di memoria fisica e la traduzione degli indirizzi a cui accede *viene conservata nel TLB.*

Se gli indirizzi della finestra di memoria non venissero invalidati quando il processore torna a livello utente, un processo attaccante potrebbe accedere speculativamente agli indirizzi di sistema acceduti dal processore, in quanto, essendo le loro traduzioni conservate nel TLB, il processore *non* utilizzerà l'albero di traduzione, aggirando così la nostra protezione contro Meltdown.

Dunque, è necessario invalidare il TLB nella funzione trampolino di uscita dal kernel (riga 565 del listato 4.9 a pagina 34).

Capitolo 4

Codice

4.1 sistema.cpp

Listing 4.1: Descrittori di processo e di TSS

```
19 // Task State Segment, richiesto dall'hardware.
20 // Per proteggere il sistema da Meltdown, e' unico per tutti
21 // i processi ed e' mappato nella memoria shadow del kernel
22 struct __attribute__ ((packed)) task_state_segment  {
23     // parte richiesta dall'hardware
24     natl riservato1;
25     vaddr punt_nucleo;
26     // due quad  a disposizione (puntatori alle pile ring 1 e 2)
27     natq disp1[2];
28     natq riservato2;
29     //entry della IST, non usata
30     natq disp2[7];
31     natq riservato3;
32     natw riservato4;
33     natw iomap_base; // si veda init_tss()
34 };
35
36 extern "C" task_state_segment tss;
37
38 // Descrittore software di processo
39 struct des_proc {
40     vaddr punt_nucleo;  // puntatore alla pila sistema del processo
41     faddr cr3;
42     natq contesto[N_REG];   // array per salvare il contesto
43     natl cpl;
44 };
45
46 // Array dei descrittori di processo
47 des_proc* descrittori_processi[NUM_TSS];
```

```
48 // Posizione immediatamente successiva all'ultimo id assegnato
49 natl next_id = 0;
50
51 // Dato un id, restituisce il puntatore al corrispondente des_proc
52 extern "C" des_proc* des_p(natl id)
53 {
54     return descrittori_processi[id];
55 }
56
57 // Cerca un'entrata libera nell'array descrittori_processi da
       ↪ assegnare
58 // al processo il cui descrittore e' passato come parametro.
59 // Restituisce l'offset dell'entrata assegnata (l'id del nuovo
       ↪ processo)
60 // oppure NUM_TSS se non ci sono piu' id liberi
61 natl seleziona_id_processo (des_proc* d)
62 {
63     natl i = next_id;
64
65     do {
66         if (descrittori_processi[i] == 0) {
67             // Trovato descrittore di processo libero
68             descrittori_processi[i] = d;
69             next_id = (i + 1) % NUM_TSS;
70             return i;
71         }
72
73         i = (i + 1) % NUM_TSS;
74     } while (i != next_id);
75
76     // Non ci sono piu' id liberi
77     return NUM_TSS;
78 }
79
80 // Rende libero l'id passato come parametro, azzerando la relativa
81 // entrata nell'array descrittori processi
82 void rilascia_id_processo (natl id)
83 {
84     descrittori_processi[id] = 0;
85 }
```

Listing 4.2: Creazione della finestra di memoria shadow

```
743 // Confini sezioni trampolino
744 extern "C" natq start_trampoline_text;
745 extern "C" natq end_trampoline_text;
746 extern "C" natq start_trampoline_data;
747 extern "C" natq end_trampoline_data;
```

```
748 extern "C" natq start_trampoline_bss;
749 extern "C" natq end_trampoline_bss;
750
751 // tab_entry che descrive la finestra di memoria shadow.
752 // Da sostituire con la finestra di memoria normale
753 // quando il programma passa a livello utente
754 tab_entry des_finestra_shadow = 0;
755
756 // tab_entry che descrive la finestra di memoria kernel completa.
757 extern "C" tab_entry des_finestra_kernel;
758
759 void mappa_pagina_shadow (vaddr ind_virt, faddr tab3, bool isText);
760
761 // Restituisce un riferimento al descrittore di livello liv
762 // da cui passa la traduzione dell'indirizzo ind_virt nella memoria
    ↪ shadow
763 tab_entry& get_des_shadow (int livello, vaddr ind_virt, faddr tab3)
764 {
765     faddr tab = tab3;
766
767     for (int i = 3; i > livello; i--) {
768         // Preleva il descrittore della tabella di livello i-esimo
769         tab_entry entry = get_entry(tab, i_tab(ind_virt, i));
770
771         if (!extr_P(entry))
772             panic("P=0 non ammesso");
773
774         tab = extr_IND_FISICO(entry);
775     }
776
777     return get_entry(tab, i_tab(ind_virt, livello));
778 }
779
780 // Se assente, crea una nuova tabella di traduzione di livello 1-3
781 // o mappa la traduzione di una pagina virtuale al livello 1
782 // nello spazio di traduzione shadow
783 void crea_shadow (vaddr ind_virt, int liv, faddr tab3, bool isText)
784 {
785     if (liv < 0 || liv > 3) {
786         panic("crea_shadow(...): valore liv non valido");
787     }
788
789     tab_entry& dt = get_des_shadow(liv + 1, ind_virt, tab3);
790     bool bit_P = extr_P(dt);
791
792     if (!bit_P && liv != 0) {
793         des_frame* df = alloca_frame_libero();
```

```
794
795         if (df == 0) {
796             flog(LOG_ERR, "Impossibile allocare copia shadow di una
                ↪ pagina");
797             panic("errore");
798         }
799
800         // Inizializza descrittore di pagina fisica
801         df->livello = liv;
802         df->residente = true;
803         df->processo = esecuzione->id;
804         df->ind_virtuale = ind_virt;
805
806         // Inizializza nuovo descrittore di tabella o pagina
                ↪ virtuale
807         faddr new_entry = indirizzo_frame(df);
808         memset(reinterpret_cast<void*>(new_entry), 0, DIM_PAGINA);
809
810         // Collega il nuovo descrittore al precedente
811         set_IND_FISICO(dt, new_entry);
812         set_P(dt, true);
813         dt |= BIT_RW;
814     }
815     else if (!bit_P) {
816         // Mappa una pagina virtuale
817         set_IND_FISICO(dt, ind_virt);
818         set_P(dt, true);
819         if (!isText) dt = dt | BIT_RW;
820     }
821 }
822
823 void mappa_pagina_shadow (vaddr ind_virt, faddr tab3, bool isText)
824 {
825     // Crea le tabelle di traduzione dal livello 2 in poi
826     for (int i = 2; i >= 0; i--) {
827         crea_shadow(ind_virt, i, tab3, isText);
828     }
829 }
830
831 // Crea la tabella di livello 3 "shadow", ovvero la FM costituita
832 // dalle pagine kernel strettamente necessario ai processi utenti
833 // e le funzioni trampolino per passare nella FM completa (ed
        ↪ uscirvi)
834 faddr crea_tab3_shadow ()
835 {
836     des_frame* df = alloca_frame_libero();
837     if (df == 0) {
```

```
838         flog(LOG_ERR, "Impossibile allocare copia shadow della
                ↪ finestra di memoria");
839         panic("errore");
840     }
841 
842     // Inizializza il descrittore di frame
843     df->livello = 3;
844     df->residente = true;
845     df->processo = esecuzione->id;
846     df->ind_virtuale = 0;
847 
848     // Inizializza la tabella
849     faddr tab3 = indirizzo_frame(df);
850     memset(reinterpret_cast<void*>(tab3), 0, DIM_PAGINA);
851 
852     return tab3;
853 }
854 
855 // Mappa nella memoria virtuale shadow
856 // la sezione text, data e bss della finestra di memoria trampolino
857 // che sono allineate a 4KiB
858 void mappa_modulo_sistema_trampolino_in_shadow (faddr tabFM)
859 {
860     natq dim_trampoline_text = (natq)&end_trampoline_text - (natq)&
            ↪ start_trampoline_text;
861     natq num_pag_text = (dim_trampoline_text / DIM_PAGINA) + 1;
862 
863     for (natq i = 0; i < num_pag_text; i++) {
864         mappa_pagina_shadow((natq)&start_trampoline_text + i*
                ↪ DIM_PAGINA, tabFM, true);
865     }
866 
867     natq dim_trampoline_data = (natq)&end_trampoline_data - (natq)&
            ↪ start_trampoline_data;
868     natq num_pag_data = (dim_trampoline_data / DIM_PAGINA) + 1;
869 
870     for (natq i = 0; i < num_pag_data; i++) {
871         mappa_pagina_shadow((natq)&start_trampoline_data + i*
                ↪ DIM_PAGINA, tabFM, false);
872     }
873 
874     natq dim_trampoline_bss = (natq)&end_trampoline_bss - (natq)&
            ↪ start_trampoline_bss;
875     natq num_pag_bss = (dim_trampoline_bss / DIM_PAGINA) + 1;
876 
877     for (natq i = 0; i < num_pag_bss; i++) {
878         mappa_pagina_shadow((natq)&start_trampoline_bss + i*
```

```
            ↪ DIM_PAGINA, tabFM, false);
879     }
880 }
881
882 void crea_finestra_FM_shadow (faddr tab4)
883 {
884     faddr tab3_shadow = crea_tab3_shadow();
885
886     set_IND_FISICO(des_finestra_shadow, tab3_shadow);
887     set_P(des_finestra_shadow, true);
888     des_finestra_shadow |= BIT_RW;
889
890     // Mappa la sezione text, data e bss della finestra di memoria
        ↪ trampolino
891     mappa_modulo_sistema_trampolino_in_shadow(tab3_shadow);
892 }
893
894 // mappa la memoria fisica in memoria virtuale, inclusa l'area PCI
895 // (copiamo la finestra gia' creata dal boot loader)
896 bool crea_finestra_FM(faddr tab4)
897 {
898     faddr boot_dir = readCR3();
899     copy_des(boot_dir, tab4, I_SIS_C, N_SIS_C);
900
901     crea_finestra_FM_shadow(tab4);
902
903     // Salva la tab_entry relativa alla finestra di memoria in
904     // modalita' kernel, nella memoria trampolino
905     des_finestra_kernel = *(reinterpret_cast<tab_entry*>(tab4));
906
907     return true;
908 }
```

Listing 4.3: Creazione di un processo

```
1009 proc_elem* crea_processo(void f(int), int a, int prio, char liv,
        ↪ bool IF)
1010 {
1011     proc_elem*  p; // proc_elem per il nuovo processo
1012     natl identifier; // identificatore del processo
1013     des_proc*   pdes_proc; // descrittore di processo
1014     des_frame*  dpf_tab4; // tab4 del processo
1015     faddr pila_sistema;
1016
1017     // ( allocazione (e azzeramento preventivo) di un des_proc
1018     pdes_proc = static_cast<des_proc*>(alloca(sizeof(des_proc)));
1019     if (pdes_proc == 0) goto errore1;
1020     memset(pdes_proc, 0, sizeof(des_proc));
```

```
1021        // )
1022
1023        // ( selezione di un identificatore
1024        identifier = seleziona_id_processo(pdes_proc);
1025        if (identifier == NUM_TSS) goto errore2;
1026        // )
1027
1028        // ( allocazione e inizializzazione di un proc_elem
1029        p = static_cast<proc_elem*>(alloca(sizeof(proc_elem)));
1030        if (p == 0) goto errore3;
1031        p->id = identifier;
1032        p->precedenza = prio;
1033        p->puntatore = 0;
1034        // )
1035
1036        // ( creazione della tab4 del processo
1037        dpf_tab4 = alloca_frame(p->id, 4, 0);
1038        if (dpf_tab4 == 0) goto errore4;
1039        dpf_tab4->livello = 4;
1040        dpf_tab4->residente = true;
1041        dpf_tab4->processo = identifier;
1042        pdes_proc->cr3 = indirizzo_frame(dpf_tab4);
1043        crea_tab4(pdes_proc->cr3);
1044        mappa_pagina_shadow(pdes_proc->cr3, extr_IND_FISICO(
               ↪ des_finestra_shadow), false);
1045        // )
1046
1047        // ( creazione della pila sistema .
1048        if (!crea_pila(p->id, fin_sis_p, DIM_SYS_STACK, LIV_SISTEMA))
1049            goto errore5;
1050        pila_sistema = carica_pila_sistema(p->id, fin_sis_p,
               ↪ DIM_SYS_STACK);
1051        if (pila_sistema == 0)
1052            goto errore6;
1053        // )
1054
1055        if (liv == LIV_UTENTE) {
1056            // ( inizializziamo la pila sistema.
1057            natq* pl = reinterpret_cast<natq*>(pila_sistema);
1058
1059            pl[-5] = reinterpret_cast<natq>(f); // RIP (codice utente)
1060            pl[-4] = SEL_CODICE_UTENTE; // CS (codice utente)
1061            pl[-3] = IF ? BIT_IF : 0; // RFLAGS
1062            pl[-2] = fin_utn_p - sizeof(natq); // RSP
1063            pl[-1] = SEL_DATI_UTENTE; // SS (pila utente)
1064            //   eseguendo una IRET da questa situazione, il processo
1065            //   passera' ad eseguire la prima istruzione della funzione
```

```
            ↪ f,
1066        //   usando come pila la pila utente (al suo indirizzo
            ↪ virtuale)
1067        // )
1068
1069        // ( creazione della pila utente
1070        if (!crea_pila(p->id, fin_utn_p, DIM_USR_STACK, LIV_UTENTE))
            ↪ {
1071            flog(LOG_WARN, "creazione pila utente fallita");
1072            goto errore6;
1073        }
1074        // )
1075
1076        // ( infine, inizializziamo il descrittore di processo
1077        //   indirizzo del bottom della pila sistema, che verra'
            ↪ usato
1078        //   dal meccanismo delle interruzioni
1079        pdes_proc->punt_nucleo = fin_sis_p;
1080
1081        //   inizialmente, il processo si trova a livello sistema,
            ↪ come
1082        //   se avesse eseguito una istruzione INT, con la pila
            ↪ sistema
1083        //   che contiene le 5 parole lunghe preparate
            ↪ precedentemente
1084        pdes_proc->contesto[I_RSP] = fin_sis_p - 5 * sizeof(natq);
1085
1086        //   il registro RDI deve contenere il parametro da passare
1087        //   alla funzione f
1088        pdes_proc->contesto[I_RDI] = a;
1089        //pdes_proc->contesto[I_FPU_CR] = 0x037f;
1090        //pdes_proc->contesto[I_FPU_TR] = 0xffff;
1091        pdes_proc->cpl = LIV_UTENTE;
1092
1093        //   tutti gli altri campi valgono 0
1094        // )
1095    } else {
1096        // ( inizializzazione della pila sistema
1097        natq* pl = reinterpret_cast<natq*>(pila_sistema);
1098        pl[-6] = reinterpret_cast<natq>(f);  // RIP (codice sistema)
1099        pl[-5] = SEL_CODICE_SISTEMA;  // CS (codice sistema)
1100        pl[-4] = IF ? BIT_IF : 0;  // RFLAGS
1101        pl[-3] = fin_sis_p - sizeof(natq);  // RSP
1102        pl[-2] = 0;  // SS
1103        pl[-1] = 0;  // ind. rit.
1104                     //(non significativo)
1105        //   i processi esterni lavorano esclusivamente a livello
```

```
1106            //    sistema. Per questo motivo, prepariamo una sola pila (
                    ↪ la
1107            //    pila sistema)
1108            // )
1109
1110            // ( inizializziamo il descrittore di processo
1111            pdes_proc->contesto[I_RSP] = fin_sis_p - 6 * sizeof(natq);
1112            pdes_proc->contesto[I_RDI] = a;
1113
1114            //pdes_proc->contesto[I_FPU_CR] = 0x037f;
1115            //pdes_proc->contesto[I_FPU_TR] = 0xffff;
1116            pdes_proc->cpl = LIV_SISTEMA;
1117
1118            //    tutti gli altri campi valgono 0
1119            // )
1120        }
1121
1122        return p;
1123
1124    errore6:    rilascia_tutto(indirizzo_frame(dpf_tab4), I_SIS_P,
                    ↪ N_SIS_P);
1125    errore5:    rilascia_frame(dpf_tab4);
1126    errore4:    dealloca(p);
1127    errore3:    rilascia_id_processo(identifier);
1128    errore2:    dealloca(pdes_proc);
1129    errore1:    return 0;
1130    }
```

Listing 4.4: Distruzione di un processo

```
1191    void distruggi_processo(proc_elem* p)
1192    {
1193        des_proc* pdes_proc = des_p(p->id);
1194
1195        faddr tab4 = pdes_proc->cr3;
1196        riassegna_tutto(p->id, tab4, I_MIO_C, N_MIO_C);
1197        riassegna_tutto(p->id, tab4, I_UTN_C, N_UTN_C);
1198        rilascia_tutto(tab4, I_UTN_P, N_UTN_P);
1199        ultimo_terminato = tab4;
1200        if (p != esecuzione) {
1201            distruggi_pila_precedente();
1202        }
1203        rilascia_id_processo(p->id);
1204        dealloca(pdes_proc);
1205    }
```

Listing 4.5: Inizializzazione del sistema

```
1838 // Azzerra tutte le entrate dell'array descrittori_processi,
1839 // rendendo liberi tutti gli id disponibili
1840 void init_descrittori_processi ()
1841 {
1842     memset(descrittori_processi, 0, sizeof(des_proc*) * NUM_TSS);
1843 }
1844
1845 void init_tss ()
1846 {
1847     //   Il campo iomap_base contiene l'offset (nel TSS) dell'inizio
1848     //   della "I/O bitmap". Questa bitmap contiene un bit per ogni
1849     //   possibile indirizzo di I/O. Le istruzioni in e out eseguite
1850     //   da livello utente verranno permesse se il bit
        ↪ corrispondente
1851     //   all'indirizzo di I/O a cui si riferiscono vale 1.
1852     //   Per disattivare questo meccanismo dobbiamo inizializzare
1853     //   il campo iomap_base con un offset maggiore o uguale
1854     //   della dimensione del segmento TSS (come scritta nel
1855     //   descrittore di segmento TSS nella GDT, vedere '
        ↪ set_entry_tss'
1856     //   in sistema.S)
1857     tss.iomap_base = DIM_DESP;
1858
1859     // Gli altri campi del TSS sono nulli
1860 }
1861
1862 extern "C" void salta_a_main();
1863 extern "C" void cmain()
1864 {
1865     natl mid;
1866
1867     // (* anche se il primo processo non e' completamente
        ↪ inizializzato,
1868     //    gli diamo un identificatore, in modo che compaia nei log
1869     init.id = 0xFFFFFFFF;
1870     init.precedenza = MAX_PRIORITY;
1871     esecuzione = &init;
1872     // *)
1873
1874     flog(LOG_INFO, "Nucleo di Calcolatori Elettronici, v5.12.6 con
        ↪ patch contro Meltdown");
1875     init_tss();
1876     flog(LOG_INFO, "tss inizializzato");
1877     init_gdt();
1878     flog(LOG_INFO, "gdt inizializzata");
1879
```

```
1880    // (* Assegna allo heap di sistema HEAP_SIZE byte nel secondo
            ↪ MiB
1881    heap_init((addr)HEAP_START, HEAP_SIZE);
1882    flog(LOG_INFO, "Heap di sistema: %x B @%x", HEAP_SIZE,
            ↪ HEAP_START);
1883    // *)
1884
1885    // ( il resto della memoria e' per i frame (parte M2)
1886    init_des_frame();
1887    flog(LOG_INFO, "Pagine fisiche: %d", N_DF);
1888    // )
1889
1890    flog(LOG_INFO, "sis/cond [%p, %p)", ini_sis_c, fin_sis_c);
1891    flog(LOG_INFO, "sis/priv [%p, %p)", ini_sis_p, fin_sis_p);
1892    flog(LOG_INFO, "io /cond [%p, %p)", ini_mio_c, fin_mio_c);
1893    flog(LOG_INFO, "usr/cond [%p, %p)", ini_utn_c, fin_utn_c);
1894    flog(LOG_INFO, "usr/priv [%p, %p)", ini_utn_p, fin_utn_p);
1895
1896    faddr inittab4 = crea_tab4();
1897
1898    if(!crea_finestra_FM(inittab4))
1899            goto error;
1900    loadCR3(inittab4);
1901    flog(LOG_INFO, "Caricato CR3");
1902
1903    apic_init(); // in libce
1904    apic_reset(); // in libce
1905    apic_fill();
1906    flog(LOG_INFO, "APIC inizializzato");
1907
1908    // ( inizializzazione dello swap, che comprende la lettura
1909    //   degli entry point di start_io e start_utente
1910    if (!swap_init())
1911            goto error;
1912    flog(LOG_INFO, "sb: blocks = %d", swap_dev.sb.blocks);
1913    flog(LOG_INFO, "sb: user    = %p/%p",
1914            swap_dev.sb.user_entry,
1915            swap_dev.sb.user_end);
1916    flog(LOG_INFO, "sb: io      = %p/%p",
1917            swap_dev.sb.io_entry,
1918            swap_dev.sb.io_end);
1919    // )
1920
1921    // ( inizializza l'array che contiene i puntatori ai descrittori
            ↪  di processo
1922    init_descrittori_processi();
1923    // )
```

```
1924
1925        // ( CREAZIONE DEL PROCESSO MAIN_SISTEMA
1926        mid = crea_main_sistema();
1927        if (mid == 0xFFFFFFFF)
1928            goto error;
1929        flog(LOG_INFO, "Creato␣il␣processo␣main_sistema␣(id␣=␣%d)", mid)
               ↪ ;
1930        // )
1931
1932        // ( CREAZIONE DEL PROCESSO DUMMY
1933        dummy_proc = crea_dummy();
1934        if (dummy_proc == 0xFFFFFFFF)
1935            goto error;
1936        flog(LOG_INFO, "Creato␣il␣processo␣dummy␣(id␣=␣%d)", dummy_proc)
               ↪ ;
1937        // )
1938
1939        // (* SELEZIONIAMO MAIN_SISTEMA
1940        schedulatore();
1941        // *)
1942        // ( ESEGUE CALL CARICA_STATO; IRETQ (VEDI "SISTEMA.S")
1943        salta_a_main();
1944        // )
1945
1946    error:
1947        c_panic("Errore␣di␣inizializzazione");
1948    }
```

4.2 sistema.s

Listing 4.6: Salva e carica stato

```
38  // OFFSET DEI VARI REGISTRI ALL'INTERNO DI DES_PROC
39  .set punt_nucleo, 0
40  .set CR3, 8
41  .set RAX,CR3+8
42  .set RCX,CR3+16
43  .set RDX,CR3+24
44  .set RBX,CR3+32
45  .set RSP,CR3+40
46  .set RBP,CR3+48
47  .set RSI,CR3+56
48  .set RDI,CR3+64
49  .set R8, CR3+72
```

```
50 .set R9, CR3+80
51 .set R10,CR3+88
52 .set R11,CR3+96
53 .set R12,CR3+104
54 .set R13,CR3+112
55 .set R14,CR3+120
56 .set R15,CR3+128
57
58
59 ////////////////////////////////
60 //      Cambio contesto       //
61 ////////////////////////////////
62
63 // copia lo stato dei registri generali nel des_proc del
64 // processo puntato da esecuzione.
65 // Nessun registro viene sporcato.
66 salva_stato:
67     // salviamo lo stato di un paio di registri
68     // in modo da poterli temporaneamente riutilizzare
69     // In particolare, useremo %rax come registro di lavoro
70     // e %rbx come puntatore al des_proc.
71     .cfi_startproc
72     .cfi_def_cfa_offset 8
73     pushq %rbx
74     .cfi_adjust_cfa_offset 8
75     .cfi_offset rbx, -16
76     pushq %rax
77     .cfi_adjust_cfa_offset 8
78     .cfi_offset rax, -24
79
80     // ricaviamo il puntatore al des_proc
81     movq esecuzione, %rax
82     movq $0, %rbx
83     movw (%rax), %bx     // Campo id del proc_elem
84     leaq descrittori_processi, %rax
85     movq (%rax, %rbx, 8), %rbx
86
87     // copiamo per primo il vecchio valore di %rax
88     movq (%rsp), %rax
89     movq %rax, RAX(%rbx)
90     // usiamo %rax come appoggio per copiare il vecchio %rbx
```

```
91      movq 8(%rsp), %rax
92      movq %rax, RBX(%rbx)
93      // copiamo gli altri registri
94      movq %rcx, RCX(%rbx)
95      movq %rdx, RDX(%rbx)
96      // salviamo il valore che %rsp aveva prima della chiamata
97      // a salva stato (valore corrente meno gli 8 byte che
98      // contengono l'indirizzo di ritorno e i 16 byte dovuti
99      // alle due push che abbiamo fatto all'inizio)
100     movq %rsp, %rax
101     addq $24, %rax
102     movq %rax, RSP(%rbx)
103     movq %rbp, RBP(%rbx)
104     movq %rsi, RSI(%rbx)
105     movq %rdi, RDI(%rbx)
106     movq %r8,  R8 (%rbx)
107     movq %r9,  R9 (%rbx)
108     movq %r10, R10(%rbx)
109     movq %r11, R11(%rbx)
110     movq %r12, R12(%rbx)
111     movq %r13, R13(%rbx)
112     movq %r14, R14(%rbx)
113     movq %r15, R15(%rbx)
114
115     popq %rax
116     .cfi_adjust_cfa_offset -8
117     .cfi_restore rax
118     popq %rbx
119     .cfi_adjust_cfa_offset -8
120     .cfi_restore rbx
121
122     retq
123     .cfi_endproc
124
125
126 // carica nei registri del processore lo stato contenuto nel
        ↪ des_proc del
127 // processo puntato da esecuzione.
128 // Questa funzione sporca tutti i registri.
129 carica_stato:
130     .cfi_startproc
```

```
131     .cfi_def_cfa_offset 8
132
133     // Carichiamo nel TSS il puntatore alla base della pila
            ↪ sistema
134     // del processo in esecuzione
135     movq esecuzione, %rcx
136     movq $0, %rax
137     movw (%rcx), %ax        // id processo
138     leaq descrittori_processi, %rbx
139     movq (%rbx, %rax, 8), %rbx  // des_proc
140     movq punt_nucleo(%rbx), %rcx
141     leaq tss, %rax
142     movq %rcx, 4(%rax)
143
144     popq %rcx   //ind di ritorno, va messo nella nuova pila
145     .cfi_adjust_cfa_offset -8
146     .cfi_register rip, rcx
147
148     // nuovo valore per cr3
149     movq CR3(%rbx), %r10
150     movq %cr3, %rax
151     cmpq %rax, %r10
152     je 1f           // evitiamo di invalidare il TLB
153                     // se cr3 non cambia
154     movq %r10, %rax
155     movq %rax, %cr3     // il TLB viene invalidato
156 1:
157
158     // anche se abbiamo cambiato cr3 siamo sicuri che
159     // l'esecuzione prosegue da qui, perche' ci troviamo
            ↪ dentro
160     // la finestra FM che e' comune a tutti i processi
161     movq RSP(%rbx), %rsp  //cambiamo pila
162     pushq %rcx  //rimettiamo l'indirizzo di ritorno
163     .cfi_adjust_cfa_offset 8
164     .cfi_offset rip, -8
165
166     // se il processo precedente era terminato o abortito la
            ↪ sua pila sistema
167     // non era stata distrutta, in modo da permettere a noi
            ↪ di continuare
```

```
168     // ad usarla. Ora che abbiamo cambiato pila possiamo
            ↪ disfarci della
169     // precedente.
170     cmpq $0, ultimo_terminato
171     je 1f
172     call distruggi_pila_precedente
173 1:
174
175     movq RCX(%rbx), %rcx
176     movq RDI(%rbx), %rdi
177     movq RSI(%rbx), %rsi
178     movq RBP(%rbx), %rbp
179     movq RDX(%rbx), %rdx
180     movq RAX(%rbx), %rax
181     movq R8(%rbx), %r8
182     movq R9(%rbx), %r9
183     movq R10(%rbx), %r10
184     movq R11(%rbx), %r11
185     movq R12(%rbx), %r12
186     movq R13(%rbx), %r13
187     movq R14(%rbx), %r14
188     movq R15(%rbx), %r15
189     movq RBX(%rbx), %rbx
190
191     retq
192     .cfi_endproc
```

Listing 4.7: Istruzioni per caricare nella GDT le primitive trampolino

```
333     carica_gate TIPO_TRAMP_IN   a_trampoline_in
            ↪ LIV_SISTEMA
334     carica_gate TIPO_TRAMP_OUT  a_trampoline_out
            ↪ LIV_SISTEMA
```

Listing 4.8: Inizializza GDT e TSS

```
364 .set p_dpl_type, 0b10001001 //P=1,DPL=00,TYPE=1001=TSS READY
365 .set pres_bit,   0b10000000
366
367     .global init_gdt
368 init_gdt:
369     lgdt gdt_pointer
370
```

```
371     // Inizilizza descrittore di TSS
372     leaq des_tss, %rdx
373     movw $DIM_DESP, (%rdx)  //[15:0] = limit[15:0]
374     decw (%rdx)
375     leaq tss, %rax
376     movw %ax, 2(%rdx)   //[31:16] = base[15:0]
377     shrq $16, %rax
378     movb %al, 4(%rdx)   //[39:32] = base[24:16]
379     movb $p_dpl_type, 5(%rdx)   //[47:40] = p_dpl_type
380     movb $0, 6(%rdx)    //[55:48] = 0
381     movb %ah, 7(%rdx)   //[63:56] = base[31:24]
382     shrq $16, %rax
383     movl %eax, 8(%rdx)  //[95:64] = base[63:32]
384     movl $0, 12(%rdx)   //[127:96] = 0
385
386     // carichiamo TR con l'offset dell'unico descrittore di
        ↪ tss
387     // (in modo che il meccanismo delle interruzioni usi la
388     // pila sistema del processo in esecuzione)
389     leaq des_tss, %rax
390     andb $0b11111101, 5(%rax)   // reset del bit BUSY
391                     // (richiesto dal processore
392                     // per compatibilita' con il modo
393                     // a 32 bit)
394     movq $(des_tss - gdt), %rax
395
396     ltr %ax
397
398     retq
```

Listing 4.9: Funzioni trampolino

```
519 ////////////////////////////////////////////////
520 //    Sezione TRAMPOLINE_TEXT: codice kernel    //
521 ////////////////////////////////////////////////
522 .section .trampoline_text, "ax"
523 .balign 4096
524 .global start_trampoline_text
525 start_trampoline_text:
526
527 // Parte in comune tra trampoline_in (chiamata dal modulo
    ↪ sistema)
```

```
528 // e a_trampoline_in (chiamata dal modulo I/O)
529 trampoline_in_2:
530     .cfi_startproc
531     .cfi_def_cfa_offset 8
532 
533     // Modifica la tab_entry della tabella di livello 4
534     // relativa alla finestra di memoria fisica,
535     // sostituendo il sottoalbero di traduzione shadow
536     // (usato per proteggere il sistema da Meltdown)
537     // con quello kernel, cosi' da poter accedere alle
        ↪ funzionalita'
538     // del nucleo
539 
540     movq des_finestra_kernel, %rbx
541     movq %cr3, %rax
542     movq %rbx, (%rax)
543 
544     retq
545     .cfi_endproc
546 
547 // Parte in comune tra trampoline_out (chiamata dal modulo
    ↪ sistema)
548 // e a_trampoline_out (chiamata dal modulo I/O)
549 trampoline_out_2:
550     .cfi_startproc
551     .cfi_def_cfa_offset 8
552 
553     // Modifica la tab_entry della tabella di livello 4
554     // relativa alla finestra di memoria fisica,
555     // sostituendo il sottoalbero di traduzione kernel
556     // con quello shadow, per proteggere il sistema da
        ↪ Meltdown.
557     // In questo modo, il programma utente non potra'
        ↪ accedere
558     // in maniera speculativa alle zone di memoria del kernel
559 
560     movq des_finestra_shadow, %rbx
561     movq %cr3, %rax
562     movq %rbx, (%rax)
563 
564     // Invalidiamo il TLB per togliere le traduzioni della
```

```
        ↪ finestra di memoria fisica
565     call    invalida_TLB
566
567     retq
568     .cfi_endproc
569
570
571 trampoline_in:
572     .cfi_startproc
573     .cfi_def_cfa_offset 8
574
575     pushq %rbx
576     .cfi_adjust_cfa_offset 8
577     .cfi_offset rbx, -16
578     pushq %rax
579     .cfi_adjust_cfa_offset 8
580     .cfi_offset rax, -24
581
582     // Controlliamo se stiamo entrando dentro il kernel dallo
        ↪  spazio shadow,
583     // controllando a quale sottospazio appartiene il %RIP
        ↪ salvato dalla
584     // chiamata dell'interruzione
585     movq 24(%rsp), %rbx     // %RIP salvato nella pila da INT
586     movabs $(1 << 47), %rax
587     addq %rbx, %rax
588     jnc 1f          // salta se non dobbiamo entrare nel
        ↪ trampolino
589
590     call trampoline_in_2
591
592 1:  popq %rax
593     .cfi_adjust_cfa_offset -8
594     .cfi_restore rax
595     popq %rbx
596     .cfi_adjust_cfa_offset -8
597     .cfi_restore rbx
598     retq
599     .cfi_endproc
600
601 // Effettua la fase di trampolino in uscita dal kernel, dopo
```

```
602 // aver controllato che il programma tornera' a livello
    ↪ utente
603 trampoline_out:
604     .cfi_startproc
605     .cfi_def_cfa_offset 8
606 
607     pushq %rbx
608     .cfi_adjust_cfa_offset 8
609     .cfi_offset rbx, -16
610     pushq %rax
611     .cfi_adjust_cfa_offset 8
612     .cfi_offset rax, -24
613 
614     // Controlliamo se stiamo passando da sistema a utente
615     movq 24(%rsp), %rbx      // %RIP salvato
616     movabs $(1 << 47), %rax
617     addq %rbx, %rax
618     jnc 1f               // salta se non dobbiamo entrare nel
        ↪ trampolino
619 
620     // STIAMO USCENDO DAL KERNEL
621     // E' necessario il trampolino in uscita
622     call    trampoline_out_2
623 
624 1:  popq %rax
625     .cfi_adjust_cfa_offset -8
626     .cfi_restore rax
627     popq %rbx
628     .cfi_adjust_cfa_offset -8
629     .cfi_restore rbx
630     retq
631     .cfi_endproc
632 
633 // Routine int $TIPO_TRAMP_IN, accessibile solo dal modulo I/
    ↪ O
634 a_trampoline_in:
635     .cfi_startproc
636     .cfi_def_cfa_offset 40
637     .cfi_offset rip, -40
638     .cfi_offset rsp, -16
639     pushq %rbx
```

```
640     .cfi_adjust_cfa_offset 8
641     .cfi_offset rbx, -16
642     pushq %rax
643     .cfi_adjust_cfa_offset 8
644     .cfi_offset rax, -24
645 
646     call trampoline_in_2
647 
648     popq %rax
649     .cfi_adjust_cfa_offset -8
650     .cfi_restore rax
651     popq %rbx
652     .cfi_adjust_cfa_offset -8
653     .cfi_restore rbx
654     iretq
655     .cfi_endproc
656 
657 // Routine int $TIPO_TRAMP_OUT, accessibile solo dal modulo I
    ↪ /O
658 a_trampoline_out:
659     .cfi_startproc
660     .cfi_def_cfa_offset 40
661     .cfi_offset rip, -40
662     .cfi_offset rsp, -16
663     pushq %rbx
664     .cfi_adjust_cfa_offset 8
665     .cfi_offset rbx, -16
666     pushq %rax
667     .cfi_adjust_cfa_offset 8
668     .cfi_offset rax, -24
669 
670     call trampoline_out_2
671 
672     popq %rax
673     .cfi_adjust_cfa_offset -8
674     .cfi_restore rax
675     popq %rbx
676     .cfi_adjust_cfa_offset -8
677     .cfi_restore rbx
678     iretq
679     .cfi_endproc
```

Listing 4.10: Esempio della parte Assembly di una routine di sistema con le chiamate alle funzioni trampolino

```
685 a_activate_p:   // routine int $tipo_a
686     .cfi_startproc
687     .cfi_def_cfa_offset 40
688     .cfi_offset rip, -40
689     .cfi_offset rsp, -16
690     call trampoline_in
691     call salva_stato
692     cavallo_di_troia %rdi
693     call c_activate_p
694     call carica_stato
695     call trampoline_out
696     iretq
697     .cfi_endproc
```

Listing 4.11: Sezioni DATA e TSS della memoria shadow/trampolino

```
1666 .global end_trampoline_text
1667 end_trampoline_text:
1668
1669 ////////////////////////////////////////////////////
1670 //   Sezione TRAMPOLINE_DATA: tabelle e dati      //
1671 ////////////////////////////////////////////////////
1672 .section .trampoline_data, "aw"
1673 .balign 4096
1674 .global start_trampoline_data
1675 start_trampoline_data:
1676
1677 // puntatori alle tabelle GDT e IDT
1678 // nel formato richiesto dalle istruzioni LGDT e LIDT
1679 gdt_pointer:
1680     .word end_gdt-gdt       // limite della GDT
1681     .quad gdt               // base della GDT
1682 idt_pointer:
1683     .word 0xFFF         // limite della IDT (256 entrate)
1684     .quad idt           // base della IDT
1685 triple_fault_idt:
1686     .word 0
1687     .quad 0
1688 param_err:
1689     .asciz "indirizzo non valido: %p"
```

```
1690 
1691 .balign 8
1692 .global gdt
1693 gdt:
1694     .quad 0     //SEGMENTO NULLO
1695 code_sys_seg:
1696     .word 0b0           //LIMIT[15:0]   NOT USED
1697     .word 0b0           //BASE[15:0]    NOT USED
1698     .byte 0b0           //BASE[23:16]   NOT USED
1699     .byte 0b10011010    //P|DPL|1|1|C|R|A|   DPL=00=SISTEMA
1700     .byte 0b00100000    //G|D|L|-|--------|   L=1 LONG MODE
1701     .byte 0b0           //BASE[31:24]   NOT USED
1702 code_usr_seg:
1703     .word 0b0           //LIMIT[15:0]   NOT USED
1704     .word 0b0           //BASE[15:0]    NOT USED
1705     .byte 0b0           //BASE[23:16]   NOT USED
1706     .byte 0b11111010    //P|DPL|1|1|C|R|A|   DPL=11=UTENTE
1707     .byte 0b00100000    //G|D|L|-|--------|   L=1 LONG MODE
1708     .byte 0b0           //BASE[31:24]   NOT USED
1709 data_usr_seg:
1710     .word 0b0           //LIMIT[15:0]   NOT USED
1711     .word 0b0           //BASE[15:0]    NOT USED
1712     .byte 0b0           //BASE[23:16]   NOT USED
1713     .byte 0b11110010    //P|DPL|1|0|E|W|A|   DPL=11=UTENTE
1714     .byte 0b00000000    //G|D|-|-|--------|
1715     .byte 0b0           //BASE[31:24]   NOT USED
1716 des_tss:
1717     .space 16,0 //SEGMENTO TSS, RIEMPITO A RUNTIME
1718 end_gdt:
1719 
1720 .global tss
1721 tss:
1722     .fill DIM_DESP, 0
1723 
1724 .global des_finestra_kernel
1725 des_finestra_kernel:
1726     .quad 0
1727 
1728 .global end_trampoline_data
1729 end_trampoline_data:
1730 
```

```
1731
1732 .section .trampoline_bss, "aw", @nobits
1733 .balign 4096
1734 .global start_trampoline_bss
1735 start_trampoline_bss:
1736
1737 // .BALIGN 16
1738 idt:
1739     // SPAZIO PER 256 GATE
1740     // VERRA' RIEMPITA A TEMPO DI ESECUZIONE
1741     .space 16 * 256, 0
1742 end_idt:
1743
1744 exc_error:
1745     .space 8, 0
1746
1747 .global end_trampoline_bss
1748 end_trampoline_bss:
```

4.3 io.s

Listing 4.12: Esempio della parte Assembly di una primitiva di I/O con le chiamate alle primitive trampolino

```
378 a_readse_n:
379     .cfi_startproc
380     .cfi_def_cfa_offset 40
381     .cfi_offset rip, -40
382     .cfi_offset rsp, -16
383     cavallo_di_troia %rsi
384     cavallo_di_troia2 %rsi %rdx
385     cavallo_di_troia %rcx
386     int $TIPO_TRAMP_IN
387     call c_readse_n
388     int $TIPO_TRAMP_OUT
389     iretq
390     .cfi_endproc
```

4.4 costanti.h

Listing 4.13: Costanti per gli id delle primitive trampoline

```
76 #define TIPO_TRAMP_IN       0x59    // TRAMPOLINE IN INGRESSO
77 #define TIPO_TRAMP_OUT      0x5a    // TRAMPOLINE IN USCITA
```

Listings

Bibliografia

[1] Graziano Frosini e Giuseppe Lettieri. *Architettura dei calcolatori Vol. II: Struttura hardware del processore PC, del Bus, della memoria, delle interfacce e gestione dell'I/O*. A cura di Pisa University Press. 2013.

[2] Graziano Frosini e Giuseppe Lettieri. *Architettura dei calcolatori Vol. III: Aspetti architetturali avanzati e nucleo di sistema operativo*. A cura di Pisa University Press. 2013.

[3] Daniel Gruss et al. «KASLR is Dead: Long Live KASLR». In: *International Symposium on Engineering Secure Software and Systems*. Springer International Publishing, 2017, pp. 161–176.

[4] Daniel Gruss et al. «Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR». In: *Conference on Computer and Communications Security*. 2016, pp. 368–379.

[5] Ralf Hund, Carsten Willems e Thorsten Holz. «Practical Timing Side Channel Attacks against Kernel Space ASLR». In: *Security and Privacy*. 2013, pp. 191–205.

[6] Yeongjin Jang, Sangho Lee e Taesoo Kim. «Breaking Kernel Address Space Layout Randomization with Intel TSX». In: *Conference on Computer and Communications Security*. 2016, pp. 380–392.

[7] Giuseppe Lettieri. *Implementazione della memoria virtuale*. 13 Mag. 2019. URL: `https://calcolatori.iet.unipi.it/resources/paginazione-nel-nucleo.pdf`.

[8] Giuseppe Lettieri. *Introduzione al sistema multiprogrammato*. 24 Apr. 2020. URL: `https://calcolatori.iet.unipi.it/resources/nucleo.pdf`.

[9] Giuseppe Lettieri. *Istruzioni all'uso del nucloe*. URL: `https://calcolatori.iet.unipi.it/istruzioni_nucleo.php`.

[10] Giuseppe Lettieri. *Memoria Cache*. 16 Mar. 2017. URL: `https://calcolatori.iet.unipi.it/resources/cache.pdf`.

[11] Giuseppe Lettieri. *Paginazione*. 3 Mag. 2019. URL: `https://calcolatori.iet.unipi.it/resources/paginazione.pdf`.

[12] Giuseppe Lettieri. *Paginazione su domanda*. 14 Mag. 2018. URL: https://calcolatori.iet.unipi.it/resources/paginazione-su-domanda.pdf.

[13] Giuseppe Lettieri. *Paginazione: complementi*. 4 Apr. 2017. URL: https://calcolatori.iet.unipi.it/resources/tlb.pdf.

[14] Giuseppe Lettieri. *Protezione*. 11 Apr. 2019. URL: https://calcolatori.iet.unipi.it/resources/protezione.pdf.

[15] Giuseppe Lettieri. *Tabelle multilivello*. 6 Mag. 2019. URL: https://calcolatori.iet.unipi.it/resources/tabelle-multilivello.pdf.

[16] Moritz Lipp et al. «Meltdown: Reading Kernel Memory from User Space». In: *USENIX Security Symposium*. 2018.

[17] Yuval Yarom e Katrina Falkner. «FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack». In: *USENIX Security Symposium*. 2014, pp. 719–732.